Redis - ESLE Report

Group 08 - Thomas Ludger (1105153)

Instituto Superior Tecnico, Lisboa, Portugal

Abstract. The goal of this work is to benchmark and analyze a distributed system to assess its behavior, scalability properties, and reason about techniques to remove or mitigate bottlenecks.
Git repository **ESLE-REDIS-8**
Git commit **68ff7d16c59f027020649780ed99263a0175ea93**

Keywords: Redis · Scalability · Benchmark · Distributed system · Database · Cache.

1 Introduction

Redis, REmote DIctionary Server, is an extensible, open-source, very high performance key-value database management system. It is part of the NoSQL movement and aims to provide the highest performance possible. Redis is very popular in the industry and is mainly used as a cache thanks to the possibility of writing data directly to RAM. It is a very versatile and complete tool with data persistence and replication options, while remaining easy to use.

1.1 Justification

We choose to study the **Redis system** for several reasons. First, Redis is an interesting system because it can have two roles, both based on performance: it can be a **key-value type memory cache**, the priority of which is the speed of the response time. But it can also serve as a more classic database for simple data. By using RAM as the main memory, it allows **quick access** to this data. On the other hand, it is a **highly scalable system**, vertically and horizontally, whose scalability properties are interesting to study and align perfectly with the rise of cloud computing. Finally, it is a system eminently present in large companies including Twitter for managing the chronology of tweets and it is a service available in the form of cloud provider services (Amazon, GCP, etc.).

2 System Description

2.1 Redis characteristics

Key-value database A key-value database or key-value store uses a simple key-value method to store data. These databases contain a simple string, the key that is always unique and an arbitrary large data field, the value. This type of NoSQL database implements a hash table to store unique keys along with the pointers to the corresponding data values. One of the advantage is that they are easy to design and implement.

Abstract data structures Redis is a data structure server and therefore manages a large number of data types such as:

- **Strings**
- **Lists**: sorted by insertion order
- **Sets**: unordered collections of unique strings that act like programming language (Java HashSets, Python sets, etc.)
- **Hashes**: record types modeled as collections of field-value pairs
- **Sorted sets**: collections of unique strings that maintain order based on the score associated with each string
- **Stream**: data structure that acts as an append-only log
- **Geospatial indexes**: useful for finding locations within a given geographic radius or bounding box
- **Bitmaps**: to perform bitwise operations on strings
- **Bitfiel**: encode multiple counters into a string value
- **HyperLog**: probabilistic estimates of the cardinality of large set

Persistence Redis stores the dataset in memory, with two different persistence methods available. The first is **snapshot**, where the dataset is asynchronously transferred from memory to disk at regular intervals as a binary dump, using the Redis RDB dump file format. The second method is **logging**: a record of each operation that modifies the dataset is appended to an append-only (AOF) file in a background process. Redis can rewrite the append file only in the background to avoid indefinite log growth. By default, Redis writes data to a filesystem at least every 2 seconds. In the event of a complete system crash to default settings, only a few seconds of data would be lost.

Replication Redis supports **master-slave** replication. Data from any Redis server can be replicated to any number of replicas. This allows Redis to implement a single root replication tree. This replication is useful for read (but not write) scalability or data redundancy.

Clustering Redis can **scale horizontally** through the functionality of Redis Cluster. It allows running a Redis installation where data is automatically shared across multiple Redis nodes. This provides some degree of availability and the ability to continue operations when some nodes fail or are unable to communicate.

2.2 System Architecture

For our experimentation, we decided to follow the instructions of the official Redis documentation. Our choice was to deploy a containerized infrastructure with the orchestration tool Docker Swarm. We created a Swarm Cluster and a volume and network for deploy our Redis cluster. Then, we created two services: one for the Redis master and one as a slave, for replication. Each service is set to only one replica for this first stage of the project.

This architecture was running on a Debian 11 virtual machine with **4 vCPU** and **8Go of RAM**.

2.3 Benchmarking strategy

To carry out the benchmark experiments, we first decided to use the tool offered by Redis, *redis-benchmark* to determine the throughput of each type of request. This benchmark performed **10,000 requests** with **50 concurrent clients** and a **data size of 3 bytes** for GET and SET requests. To deepen our study, we decided to use a well-known database benchmarking tool: **Yahoo! Cloud Serving Benchmark (YCSB)** to determine the **throughput** and the **latency** for each type of request. It is natively compatible with Redis and offers workload templates. We modified our workload for these characteristics:

- 50% of READ requests,
- 20% of SCAN,
- 15% of UPDATE,
- 15% of INSERT
- data size of 1KB (10 fileds, 100 bytes each)
- 1000 target operations over a data set of 1000

We ran this experiment several times, varying the number of threads (concurrent clients) from 1 to 100, with a step of 5.

Alongside the YCSB benchmark, we analyzed CPU and memory usage, thanks to Docker stats.

3 First results

3.1 Redis Scalability

To analyze the results of the benchmarks, we made different plots, with the *gnuplot* tool. The results of our first experiment with the *redis-benchmark* tool, shown in *Figure 2*, showed us a throughput of 6,000 to 24,000 requests per second, depending on the type of request.

Our second experiment with YCSB, plotted in *Figures 3 and 4*, showed us a very high and rather odd SCAN-like write latency of up to 225ms during 100 concurrent clients. The difference results between YCSB and *redis-benchmark* can be explained with the data size difference (1KB cs 3KB) and the different type of requests (ex. *get* for Redis-benchmark and *hmset* for ycsb).

We also plotted the throughput (*Figure 5*) revealed by this experiment and we used the same data to determine the parameters of the **Universal Scalability Model** (1) equation using the *esle-usl-1.0-SNAPSHOT.jar* tool:

$$X(N) = \frac{\lambda N}{1 + \delta(N-1) + \kappa N(N-1)} \tag{1}$$

The tool gives us these parameters results:

$$\lambda = 418,36 \quad \delta = 0,4 \quad \kappa = 0,00074 \tag{2}$$

Finally, the analysis of the use of the CPU and the memory (figure 6 and 7) showed us slight variations of the CPU according to the number of threads, in particular for the CPU which can go up to 79%. Memory is not excessively stressed.

3.2 Stage Pipeline

To find a potential bottleneck and deepen our understanding of the Redis system, we tried to break down the workload requests into stages using the logs and also the official documentation in order to build a typical pipeline.

We did the two diagrams Figure 8 and Figure 9 from our understanding to explain the global working of Redis for a *Get* and *Set* request and the replication and persistence features.

3.3 First bottleneck identification and mitigation

Our multiple experiments reveal a possible bottleneck at the CPU level which quickly reaches its limits during the benchmarks.

Another possible bottleneck would be the network, which according to the Redis documentation can quickly be the main bottleneck.

To counter this problem, it is recommended to use the **pipelining** feature which groups several commands into one. We will try to determine this to improve performance.

4 Experimental design

4.1 Metrics, factors and levels

To evaluate the performance of our Redis system, we decided to study throughput (requests per second) and latency (response time) metrics. The selected factors and their respective levels are:

Redis Cluster Redis scales horizontally with the ability to deploy a Redis Cluster. This allows for a Redis infrastructure where data is automatically shared across multiple Redis nodes. Thus, Redis availability is improved with the ability to continue operations when some nodes fail or are unable to communicate. The dataset is therefore split between multiple nodes and operations can continue even if one node experiences a failure or is unable to communicate with the rest of the cluster.

So, we can establish two factors regarding clustering:

– **Level -1**: standalone mode (non-cluster)
– **Level 1**: cluster with 3 masters and 3 slaves

Persistence : It refers to writing data to durable storage, such as a solid-state drive (SSD).

– **Level -1**: no persistence
– **Level 1**: AOF persistence, which logs every write operation received by the server. These operations can then be replayed again at server startup, reconstructing the original dataset.

overcommit_memory value : The kernel virtual memory acounting mode, defined by the overcommit_memory value on the host, which is on 0 by default. By changing to 1, the kernel pretends there is always enough memory, until memory actually runs out.

– **Level -1**: overcommi_memory=0
– **Level 1**: overcommit_memory=1

Pipelining : Redis pipelining is a technique for improving performance by issuing multiple commands at once without waiting for the response to each individual command.

– **Level -1**: no pipelining
– **Level 1**: pipelining

Type of CPU : We saw with our first experiments that the CPU could be a bottleneck so we will do the experiments on two differents types of CPU d'une instance Google Cloud Platform.

– **Level -1**: e2-standard-8, 8vCPU
– **Level 1**: e2-highmem-16, 16vCPU

Replication : use of master-slave architecture for read-only replication

– **Level -1**: Replication to a slave Redis server
– **Level 1**: No replication, no slave Redis server

4.2 Cloud Setup

To carry out the experiments of the various factors presented before, we decided to continue our study in the Cloud, precisely on Google Cloud Platform. We deployed two virtual machine instances:

– e2-standard-8: 8vCPUS, 16GB of RAM, europe-west9-a zone, Debian 11 image
– e2-highmem-16: 16vCPUS, 64GB of RAM, europe-west9-a zone, Debian 11 image

In the machines, we re-configured a Docker Swarm as well as a Redis service.

4.3 Experiment details

We used the **Fracitonal Factorial Design** method to do only 8 experiments with our 6 factors with 2 levels design. We did one iteration for each experience. We organize the results of our experiences with two tables (see annexes):

– **Table 1** presents the latency and throughput results of the experiences for READ and WRITE command (respectively get and set)
– **Table 2** presents the sign table, the experiments configurations and the impact percentages of each factor for each metric.

The benchmark was done using the included Redis tool *redis-benchmark* with 1000 requests of 1KB data and only for the GET and SET type of requests. For pipelining, we use the pipelining option of this tool with 15 requests/pipeline. Regarding the number of thred, we choose to take experiments values of 21 threads because the first stage shows us that it s approximately with 21 threads that we have the best performances.

With these experiments, we can determine that the most important contributing factor is confounding Pipelining (D) with Clustering (A) and Persistence (B). When this option is activated, it has an impact of more than 95% on the throughput whether it is in writing or in reading and also with an impact of more than 47% in reading and 72% in writing on latency. The other factors with the most impact on read latency is Clustering with 18%. Then, the Persistence option has an impact of 13% on read latency and 10% on write latency. Another interesting impact is the overcommit setting and the type of CPU combined with Clustering and Overcommit with an impact of 6.52% on the write latency.

4.4 Scalability and Performance

From our experiences and the Redis documentation, the best performance is when the settings are:

- Cluster mode
- Persistence enabled
- overcommit_memory=1
- Pipelining enabled
- Replication enabled

We were able to determine the theoretical equation in the same way as in stage 1 with the tool *esle-usl* and thanks to the data of experiment 8 which had very good performances (reach up to 500000req/s in read and 330000 in write). We plotted it using *GNUplot*(see the Figure 10).

The parameters of the Universal Scalability Law determined are the following:

Read :

$$\lambda = 201000 \quad \delta = 0,51 \quad \kappa = 0,00033 \tag{3}$$

Write :

$$\lambda = 196529,8 \quad \delta = 0,77 \quad \kappa = -0,0007 \tag{4}$$

5 Conclusion

The study of the Redis system allowed us to familiarize ourselves with the concept of distributed systems as well as the performance issues to ensure correct scalability. By analyzing the characteristics and performance of Redis, through multiple tests, we were able to determine the three factors related to the **Universal Scalability Law**. We were also able to understand the pipeline stages of a redis request.

Finally, through the fractional factorial experimental design, we were to test 6 factors and their impact on the Redis performances system. We were able to determine that the pipelining contributing factor is a really efficient feature that have the most impact on the performance system. It allows us to analyse the Redis sacalability properties with more details and evaluate more precisely the performance coefficient, the serial portion and the crosstalk factor for Read and Write operations.

Based on the properties and behavior of the system analyzed, we can definitely spot great qualities and a robust architecture. However, there are still room to improvement, as we could show though the report.

6 Annexes

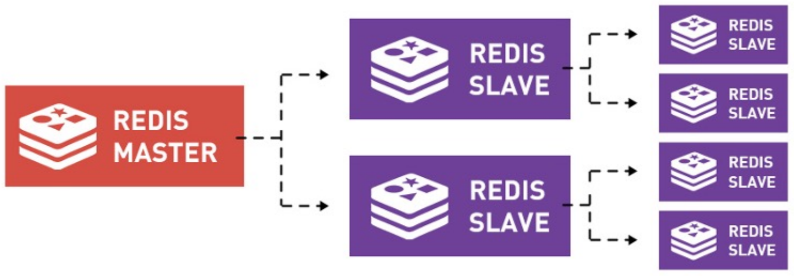

Fig. 1. Redis master-slave replication

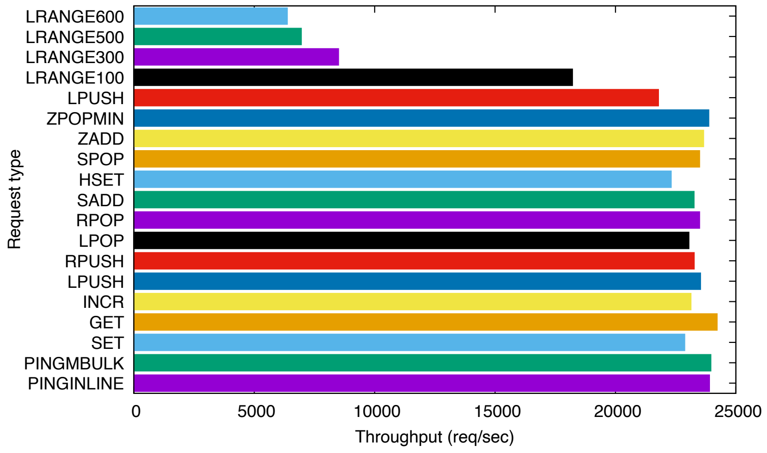

Fig. 2. Throughput plot (redis-benchmark)

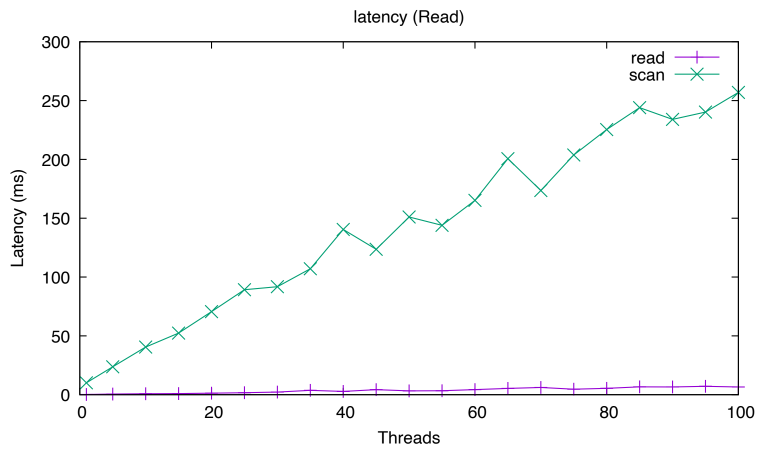

Fig. 3. Read latency plot (YCSB)

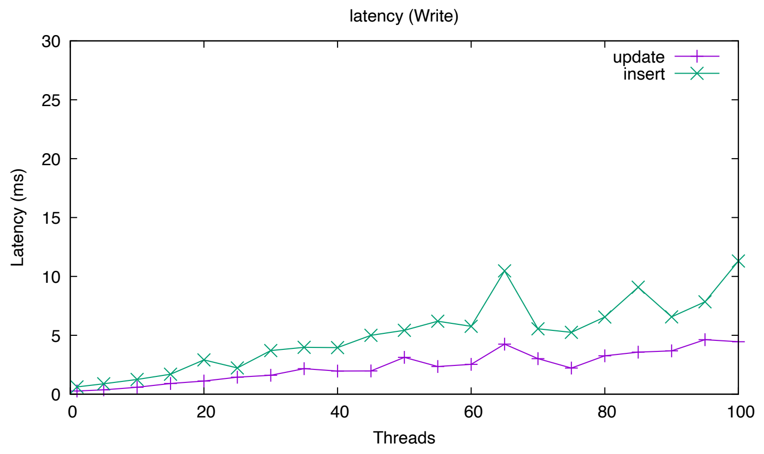

Fig. 4. Write latency plot (YCSB)

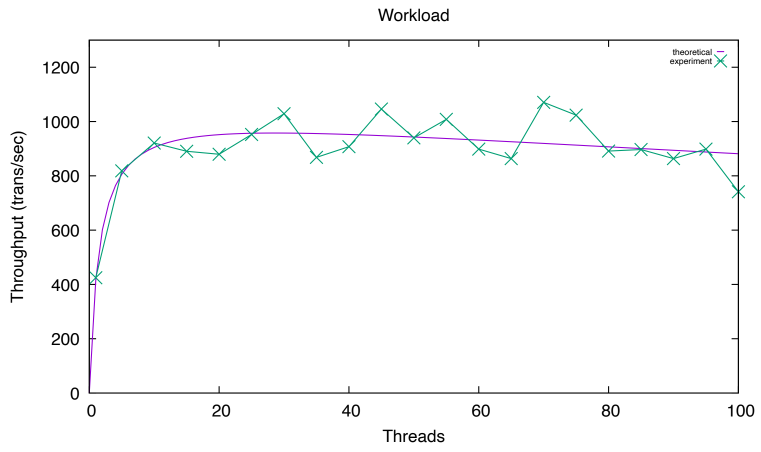

Fig. 5. Throughput plot (YCSB)

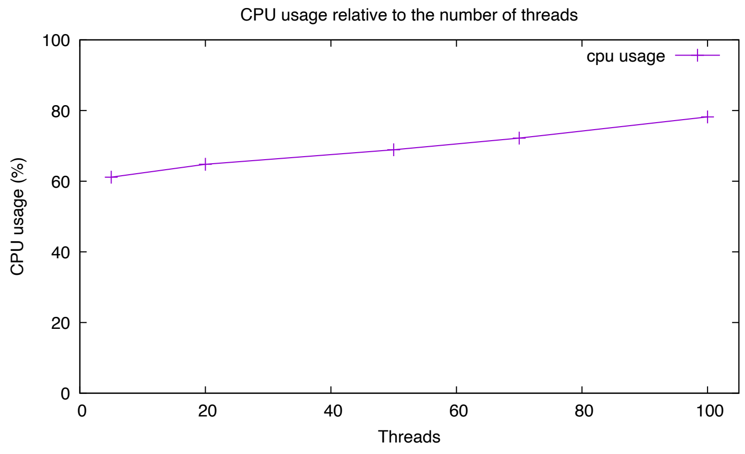

Fig. 6. CPU usage during YCSB benchmark

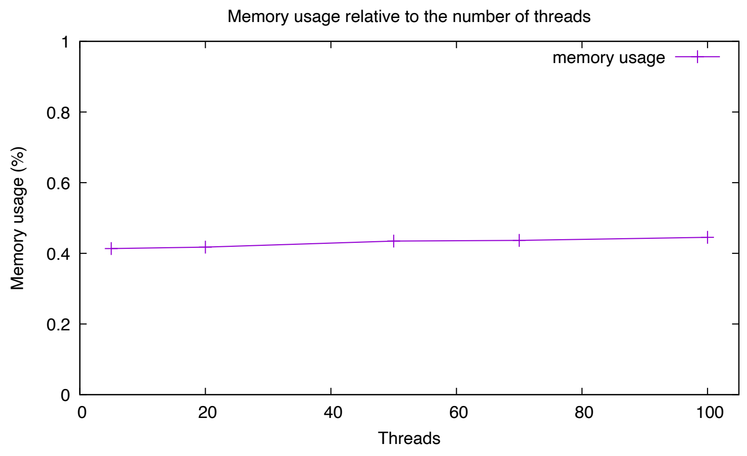

Fig. 7. Memory usage during YCSB benchmark

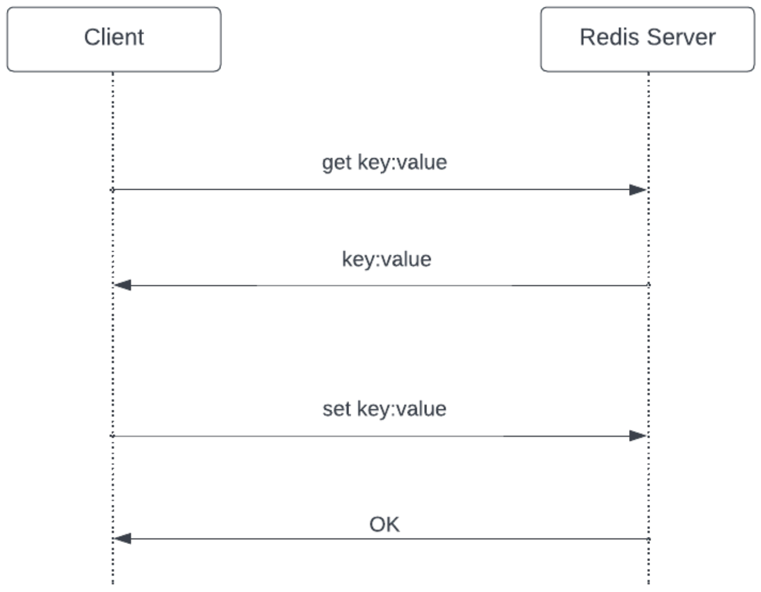

Fig. 8. Request diagram of a Redis request

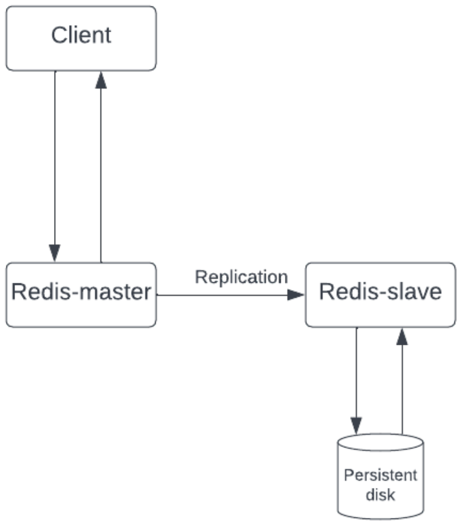

Fig. 9. Persistence and Replication of Redis

Table 1. Experiments results (Cloud)

Experience	Mean latency (μs)($get - set$)	Mean throughput (req/s) (get-set)
1	397 - 612	502500 - 335000
2	237 - 293	50000 - 38461
3	334 - 319	35714 - 32258
4	395 - 1310	502499 - 251249
5	469 - 811	502500 - 335000
6	286 - 287	50000 - 55555
7	430 - 384	34483 - 37037
8	451 - 882	502499 - 335000

Table 2. Experiments results and impact of each factor for each metric

	Experiment	Clustering	Persistence	Overcommit	Pipelining	CPU	Replication
		A	**B**	**C**	**D (AB)**	**E (AC)**	**F (BC)**
	1	-1	-1	-1	1	1	1
	2	1	-1	-1	-1	-1	1
	3	-1	1	-1	-1	1	-1
	4	1	1	-1	1	-1	-1
	5	-1	-1	1	1	-1	-1
	6	1	-1	1	-1	1	-1
	7	-1	1	1	-1	-1	1
	8	1	1	1	1	1	1
READ							
Latency (μs)	2999	-261	221	273	425	-63	31
Throughput (req/s)	2180195	29801	-29805	-1231	1839801	1231	-1231
WRITE							
Latency (μs)	4898	646	892	-170	2332	-698	-556
Throughput (req/s)	1419560	-59030	-108472	105624	1092938	96066	71436
MEAN - READ							
Latency (μs)	374.88	-32.63	27.63	34.13	53.13	-7.88	3.88
Throughput (req/s)	272524.38	3725.13	-3725.63	-153.88	229975.13	153.88	-153.88
MEAN - WRITE							
Latency (μs)	612.25	80.75	111.5	-21.25	291.5	-87.25	-69.5
Throughput (req/s)	177445	7378.75	-13559	13203	136617.25	12008.25	8929.5
READ							
SS Latency (μs)	47143.5092	8517.7352	6107.3352	9318.8552	22582.3752	496.7552	120.4352
SS Throughput	423331106945	111012748.1	111042551.2	189432.4352	423108483348	189432.4352	189432.4352
WRITE							
SS Latency (μs)	934553.5	52162.5	99458	3612.5	679778	60900.5	38642
SS Throughput	154406549421	435567612.5	1470771848	1394553672	149314183981	1153584545	637887762
READ							
% Latency (μs)	100%	18.07%	12.95%	19.77%	47.90%	1.05%	0.26%
% Throughput	100%	0.025%	0.025%	0.00%	99.95%	0.00%	0.00%
WRITE							
% Latency (μs)	100%	5.58%	10.64%	0.39%	72.74%	6.52%	4.13%
% Throughput	100%	0.29%	0.95%	0.90%	96.70%	0.75%	0.41%

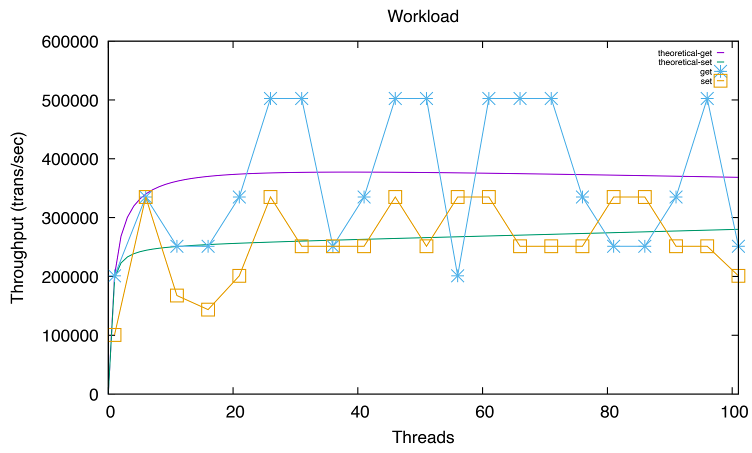

Fig. 10. Read and Write best throughput performances

References

1. Redis Benchmark tool, https://redis.io/docs/management/optimization/benchmarks/
2. Redis characteristics, https://redis.io/docs/about/
3. Redis Data Types, https://redis.io/docs/data-types/
4. Pipelining, https://redis.io/docs/manual/pipelining/
5. Definition, https://aws.amazon.com/fr/elasticache/what-is-redis/